PYBUS – A PYTHON SOFTWARE BUS

W. T. L. P. Lavrijsen*, LBNL, Berkeley, CA 94530, USA

Abstract

A software bus, just like its hardware equivalent, allows for the discovery, installation, configuration, loading, unloading, and run-time replacement of software components, as well as channeling of inter-component communication. Python, a popular open-source programming language, encourages a modular design on software written in it, but it offers little or no component functionality. However, the language and its interpreter provide sufficient hooks to implement a thin, integral layer of component support. This functionality can be presented to the developer in the form of a module, making it very easy to use. This paper describes a Python module, PYBUS, with which the concept of a "software bus" can be realized in Python. It demonstrates, within the context of the ATLAS software framework ATHENA, how PYBUS can be used for the installation and (run-time) configuration of software, not necessarily Python modules, from a Python application in a way that is transparent to the end-user.

INTRODUCTION

The software in use and being written today for the next generation of High Energy Physics (HEP) detectors is so large, that link times have become prohibitively long. The straightforward, and generally applied, solution is to make use of dynamic linking, which gives rise to a new problem: making sure that all the required shared libraries are located and properly setup for use. This situation is further complicated in an interactive environment, where the user wants to build and specify new modules on the fly, effectively working with them in a similar way as with hardware components.

In ATLAS [1] and in LHCb [2], two of the Large Hadron Collider (LHC) [3] experiments, the interactive software environment is provided by the Python interpreter [4], which has only limited support for locating and (un)loading of modules. PYBUS, itself implemented as a Python module, improves on this by acting as a software bus, which allows for the discovery, installation, configuration, loading, unloading, and run-time replacement of software components.[1]

In this paper, the PYBUS architecture and the underlying ideas, as well as its implementation are described.

* WLavrijsen@lbl.gov

[1] In this paper, no distinction will be made between modules, (pure) Python modules, and software components

ARCHITECTURE

The four basic design principles for a software bus are defined in [5]:

- **Minimal core semantics.** The bus becomes an effective communication device by minimizing the requirements on communicating applications.
- **Self-describing objects.** Introspection can be used to translate objects from one type into another, to ease communication between applications, as well as to determine interest in the objects among listeners.
- **Dynamic classing.** New, or modified, functionality can be made available without taking the bus "down," by implementing new, or modifying existing, classes dynamically.
- **Anonymous communication.** Data objects are sent and received in a space, time, and synchronization decoupled manner, and independent of the identities and data producers and data consumers, as illustrated in Fig. 1.

Where the authors of [5] had to implement their own programming language to achieve the above, Python practitioners will recognize immediately that much of the needed functionality is provided out-of-the-box by the Python interpreter. In particular, dynamic classing and the availability of reflection information are an essential part of the Python language. Further, the pieces that are missing, are mostly already there in a rudimentary form. Thus, PYBUS can be implemented as a light-weight layer (in fact, as a Python module), that interacts naturally with the Python interpreter.

The choice of PYBUS as a module makes life easier for the end-user, but it does set some "play nice" rules for modules and the use of modules. That is, PYBUS is a client of the interpreter, just like any other module, and has no special privileges. It is important to note that because of this flat structure, PYBUS-style components are normal Python modules: the software bus only adds to the ways that they can be used and there are thus no restrictions on normal Python uses.

IMPLEMENTATION

The core of the implementation consists of improvements on already existing Python functionality, in order to deal with more complex situations. The following is a list of component functionality made available by PYBUS.

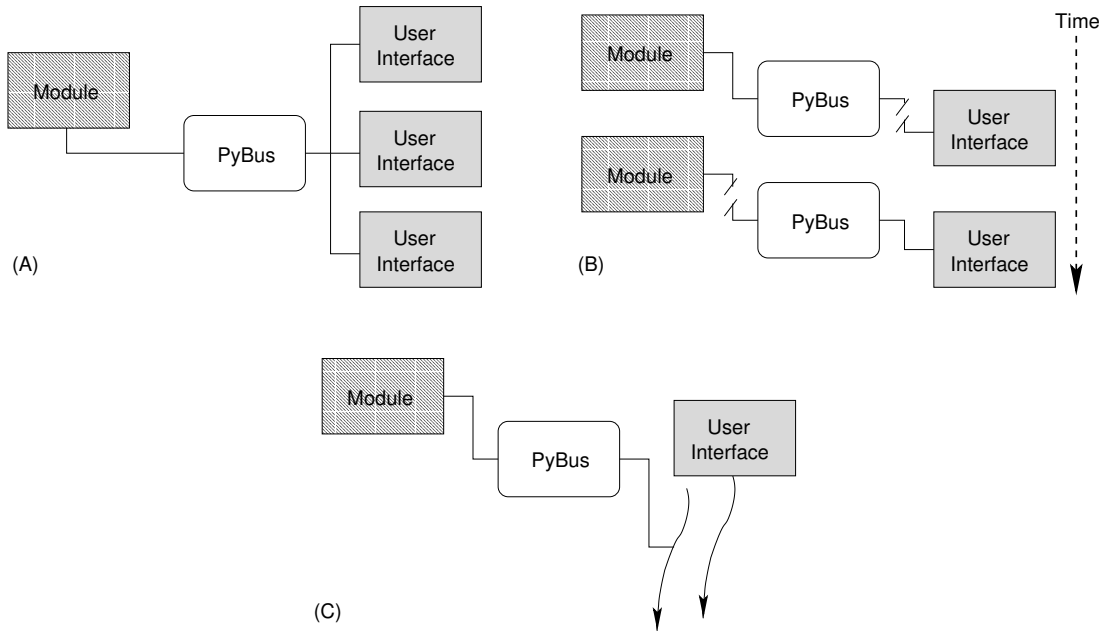

Figure 1: A software bus anonymizes communications to allow for space decoupling, e.g. multiple interfaces using the same module (A). Time decoupling, e.g. an interface can use data produced by a module without being connected at the same time (B). And synchronization decoupling (C), e.g. the interface is notified through a callback when the work done by a module is finished, thus, it need not block. [6]

Symbolic Component Names and Lookup

Python modules are loaded on the basis of names that map directly to the filesystem; components are loaded according to the functionality that is requested. A request for functionality that is known by PyBus is followed by localizing the module that contains a component with that functionality, and performing the appropriate steps, which may include downloading the component from a cache and installing it, to load it into the Python interpreter.

When a user requests a component to be connected, she can do so with the logical, functional, or actual name. Components that are available to PyBus must be registered first under logical names, optionally advertising under functional names the public contracts that they fulfill (their "interfaces" in a sense). Note that registered does not mean that it has to be available: upon a request to connect a module, it may need to be retrieved or configured first.

If a component is not registered, it can only be connected using its actual name, which is the name that would be used in the standard way of identifying a Python module. Unlike the actual name, which has to be unique, the logical name and functional names may be claimed by more than one component. PyBus will choose among the available components on the basis of its own configuration, a priority scheme, or a direct action from the user.

Connecting, Disconnecting and Replacing

Connections that are based on symbolic names allow the application writer to integrally replace a (version of a) component throughout the application at startup. PyBus adds the capability to search through outstanding references to replace or remove these individually once the component to which they point gets replaced, or removed, at run-time.

The replacement of modules is achieved by using the garbage collection and reference-counting information of the Python interpreter to track down any outstanding references, and to act accordingly: some references, e.g. those to variables or instances, are rebound, whereas others, e.g. object instances, are destroyed. Disconnecting a component is rather similar to replacing it, with the only exception that no references are rebound: all are destroyed.

Configuration and Dependencies

A PyBus-enabled module advertises configuration parameters and dependencies that can subsequently be externalized and managed globally. This allows for integral, site specific environment setups.

In the process of connecting a module, PyBus will look for conventional parameters (starting with "PyBus_") in the dictionary of the module that contains the component. These parameters may describe dependencies, new components to be registered, post-connect or pre-disconnect configuration, and so on. It is purely optional for a module to provide any of these parameters and PyBus will use some heuristics if they are absent. For example, all public identifiers are considered part of the interface, so that any module can be connected as if it were a component. The user can decide the name under which the module should be connected, which can be any alias that stands in for the name

to be used in the user code, similar to the "as" option for standard Python imports. If no alias is provided, the logical name is used. PyBus keeps track of the aliases.

Python allows user modules to intercept the importing of other modules, by replacing the import hook. This mechanism allows the PyBus implementation to bookmark those modules that are imported during the process of connecting a component, and thus manage component dependencies. When disconnecting a component, the modules that it loaded are not automatically removed, since the interpreter itself holds on to a reference to them, even after all user-level modules are unloaded. There is no real reason to force the unloading of standard modules, but if the modules are components that are connected to PyBus, they are unloaded, too.

Note that when a new component is loaded which in turn loads other components, PyBus needs to resolve the lookup of those components anew: the registration for these dependent components may have changed and different actual components may be chosen this time around.

User Interface Presentation Layer

The communication between components is anonymized using publish/subscribe, see [6], and formalized with adapters to ensure that proper change notification events are sent out, and to allow components to specify the preferred/optimal representation in a user interface (for adapters used in communication by software components, see e.g. [7] and [8]).

The adapters form a User Interface Presentation Layer (UIPL), through which the configuration, input and output parameters, and functionality of components can be connected to user interface elements. The bus inspects the component for presentable elements, including (if applicable) their type, range, name, and documentation. It subsequently requests the user interface to supply elements that are capable of providing a display of and/or interaction with each of the parameters, based on their type, range, etc. Both the interface element and the component should then be hooked through the UIPL.

For example, assume that a configuration parameter of a component is of a boolean type. This parameter can then map onto e.g. a check box in a GUI. The bus requests the GUI to provide a display of the boolean value, gets a check box in return, and it subscribes the check box to a value holder in the UIPL. It also subscribes itself to this holder. Changes by the check box, changes the value in the value holder, which in turn causes a notification to the bus, which sets the value in the component.

Mapping through an UIPL has the advantage that simple user interfaces can be created automatically, and more sophisticated user interfaces can be relatively easily peeled off and replaced, since they never access the actual underlying component directly.

Miscellaneous

In addition, modules for various ways of discovery and installation of components, and for compatibility between Python versions are provided with the implementation. For example, a given directory location can be scanned and any modules found automatically registered for subsequent connection.

EXAMPLE

A relatively straightforward example is given in Fig. 2, where the configuration tool for the ATLAS run-time environment is implicitly used to setup access to a requested module.

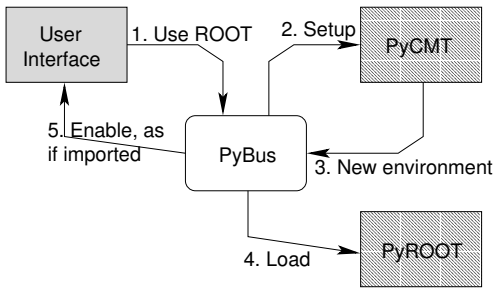

Figure 2: PyBus example: setting up PyROOT in the ATLAS environment.

First, the user requests the ROOT module (1), for which PyBus finds in its configuration that it needs to setup the run-time environment (2). A new environment is set up (3), and used to locate PyROOT and to load it into the Python interpreter, while pre- and post-configuration are handled as required (4). Finally, the module is "injected" into the user module namespace as if it was imported and can now be used (5).

OUTLOOK

PyBus is currently used in the Athena framework to handle some incompatibilities between Python versions 2.2 and 2.3. Also, the GANGA project [9] intends to use PyBus for configuration of user provided handlers.

REFERENCES

[1] Atlas Collaboration, "ATLAS – Technical Proposal", CERN/LHCC94-43, CERN, December 1994.

[2] LHCb Collaboration, "LHCb – Technical Proposal", CERN/LHCC98-4, CERN, February 1998.

[3] LHC Study Group, "The LHC conceptual design report", CERN/AC/95-05, CERN, October 1995.

[4] G. van Rossum and F. L. Drake, Jr. (eds.), "Python Reference Manual", Release 2.3.4, PythonLabs, May 2004. http://www.python.org.

[5] B. Oki, *et al.*, "The Information Bus. An Architecture for Extensible Distributed Systems", ACM 0-89791-632-8/93/0012, December 1993.

[6] P. Th. Eugster, *et al.*, "The Many Faces of Publish/Subscribe", ACM 0360-0300/03/0600-0114, June 2003.

[7] N. Carriero, D. Gelernter, "Linda in context", ISSN: 0001-0782, April 1989.

[8] S. Lewis, "The Art and Science of SmallTalk", ASIN: 0133713458, Prentice Hall, May 1995.

[9] A. Soroko, *et al.*, "The GANGA user interface for physics analysis on distributed resources", CHEP 2004, Conference Proceedings.